UCRL-TR-216197

LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

Performance of Rank-2 Fortran 90 Pointer Arrays vs. Allocatable Arrays

E. Zywicz

October 11, 2005

An informal report for communication of compiler performance issues with Livermore Computing vendors.

This work was performed under the auspices of the U.S. Department of Energy by University of California, Lawrence Livermore National Laboratory under Contract W-7405-Eng-48.

UCRL-TR-216197

Performance of Rank-2 Fortran 90 Pointer Arrays vs. Allocatable Arrays

Edward Zywicz

October 11, 2005

Methods Development Group
Lawrence Livermore National Laboratory
PO Box 808, L-125
Livermore, CA 94551

Introduction

The computational performance of two-dimensional Fortran 90 arrays defined with the `pointer` attribute were compared to identically sized arrays defined with the `allocatable` attribute. The goal of this work was to quantify the computational cost of using each array type within a high-performance finite element setting.

Test Program

A test program was developed that mirrors how the main 2-D arrays are employed within the explicit finite element code DYNA3D. The test code first allocates the array (3 by 1,000,000) and then calls a subroutine 1000 times with the array and the array size as its calling arguments. The CPU time necessary to complete all 1000 calls was measured. The Fortran coding for each variant tested was simply:

```
call cpu_time(start)
do i=1,ntimes
  call try_ia_work(ap,size)
enddo
call cpu_time(finish)
time = finish-start
```

Two types of subroutines were tested. The “working” subroutine type performs operations on the array and explores both the cost of passing the array to the subroutine as well as the cost to operate on it. The main sections of these subroutines contain the coding:

```
do i=1,size
  a(1,i) = one
  a(2,i) = two
  a(3,i) = three
enddo

do i=1,size
  a(3,i) = a(1,i) + a(2,i)
enddo
```

The “passing” type of subroutines merely adds the first two numbers of the array together and returns. It is intended to explore the cost of passing the array to the subroutine, and their main sections look like:

```
a(1,1) = one
a(2,1) = a(1,1) + one
```

Various combinations of array types, subroutine types, subroutine interfaces, and calling statements were explored. In the main calling routine, the main array had one of two attributes - `allocatable` or `pointer`. In the subroutine, the main array had a declared size and, in some cases, was assigned the `pointer` attribute. In general, the array was passed with an implicit range (`call try(a,nsize)`); however, in some cases, the array range was explicitly specified (`call try(a(1:3,1:Nsize))`). When admissible, both explicit and implicit interfaces were tested; when the array in the subroutine was assigned the `pointer` attribute, only explicit interfaces are permitted. Both working and passing types of subroutines were tested for all permutations. Table 1 summarizes the 14 different combinations considered.

Table 1: Permutations Examined

Case #	Main Routine - Array Attribute	Subroutine - Array Attribute	Subroutine Type	Interface	Range Specified
1	`Allocatable`	None	Working	Implicit	No
2	`Allocatable`	None	Passing	Implicit	No
3	`Allocatable`	None	Working	Explicit	No
4	`Allocatable`	None	Passing	Explicit	No
5	`Pointer`	None	Working	Implicit	No
6	`Pointer`	None	Passing	Implicit	No
7	`Pointer`	None	Working	Explicit	No
8	`Pointer`	None	Passing	Explicit	No
9	`Pointer`	None	Working	Implicit	Yes
10	`Pointer`	None	Passing	Implicit	Yes
11	`Pointer`	None	Working	Explicit	Yes
12	`Pointer`	None	Passing	Explicit	Yes
13	`Pointer`	`Pointer`	Working	Explicit	No
14	`Pointer`	`Pointer`	Passing	Explicit	No

Contained in the Appendix is the full source code listing for the test program.

Results

The program was compiled and run on four different platforms using their default Fortran compiler. The four platforms tested were:

1) Krakov (SGI R14,000) using f90 (version 7.4.2m) with –O3 –mips4 -64,
2) ILX (Intel Xeon) using ifort (version 8.0) with –O3,
3) GPS (Compaq Tru64) using f90 with -fast,
4) uP (IBM Power5) using xlf90 with –O4.

Table 2 contains the average CPU times from three runs on each platform.

Table 2: CPU times (seconds – average of 3 runs)

Case #	SGI - Krakov	Intel Xeon - ILX	Compaq - GPS	IBM Power5 - uP
1	40.5	25.3	18.9	1.14
2	1.21e-4	0	0	0
3	40.6	26.4	19.0	1.16
4	1.24e-4	0	0	0
5	40.5	26.0	18.9	1.16
6	1.31e-4	0	0	0
7	40.5	27.34	18.9	1.16
8	1.57e-4	0	0	0
9	168	25.6	18.9	5.88
10	129	0	0	4.77
11	169	25.4	18.9	5.85
12	129	0	0	4.79
13	36.2	24.8	25.8	3.53
14	1.32e-4	0	0	0

Analysis of Results

SGI platform:

The results for the SGI suggest that as long as you do not explicitly declare the array range on the calling line (cases 9-12), the performance is not significantly impacted by what attributes an array has or how it is passed to a subroutine (cases 1-8; 13-14). However, arrays declared with the pointer attribute and passed with an explicit interface (cases 13-14) perform slightly better, ~10% faster. Given the drastic increase in CPU time when the array ranges are explicitly specified on the calling line (cases 9-12), one might assume that a "copy-in/copy-out" operation is being performed. Clearly, specifying the array ranges should be avoided when possible.

Intel Xeon platform:

The performance on the Intel platform is completely independent of the coding style or attributes used. Unfortunately, the averaged results summarized in Table 2 are somewhat misleading due to machine utilization. Often, one of the three values averaged was 15% to 30% larger than the other two, and this variation causes the averaged values to vary a bit. However, based upon the minimum time for each case, no appreciable difference exists between the cases examined.

Compaq Tru64 platform:

There is no difference in performance with coding style or attributes used, except when the array has the pointer attribute in both the main routine and the subroutine (cases 13-14). In the later case there is a marked degradation in performance (~37%).

IBM Power5 platform:

On the IBM, as long as you do not explicitly declare the array range on the calling line (cases 9–12) or assign the pointer attribute to the array in the subroutine (cases 13-14), the performance is independent of the coding style and array attributes used in the main routine (cases 1–8). Like the SGI, arrays passed with their ranges explicitly declared are appreciably slower, and this suggests that a “copy-in/copy-out” operation is occurring. Furthermore, like the Compaq, a substantial slow down (3 times) occurs when the array in the subroutine is assigned the pointer attribute (cases 13-14).

Conclusions

In general, explicitly declaring the array ranges in a call statement should be avoided since it can cause a substantial degradation in performance on some platforms (e.g., SGI and IBM Power5) and offers no performance benefits. Unfortunately, based upon this work, no conclusions can be drawn about which array attributes are best to use within a finite element framework due to vendor variations in the Fortran compilers. The consistent use of arrays with the pointer attribute (i.e., in the main and all subsequent subroutines) show marked degradation on select platforms. For example, on the IBM they are 3 times slower other attribute combinations. This is a shame since the consistent utilization of pointer arrays is highly desirable within a finite element code, i.e., they do not have the same language limitations imposed on them as allocatable arrays.

Appendix

```
module type_vars
        implicit none

        integer, parameter:: singR = kind(0.)
        integer, parameter:: fullR = kind(0.d0)
        integer, parameter:: singI = kind(0)
        integer, parameter:: fullI = selected_int_kind(18)
        real(fullR), parameter :: ZERO     = 0.0_fullR
        real(fullR), parameter :: ROOT_EPS = 0.0000000596046448_fullR ! 2^(-24) =
5.96046448E-8
        real(fullR), parameter :: ONE      = 1.0_fullR
        real(fullR), parameter :: TWO      = 2.0_fullR
        real(fullR), parameter :: THREE    = 3.0_fullR
        real(fullR), parameter :: FOUR     = 4.0_fullR
        real(fullR), parameter :: FIVE     = 5.0_fullR
        real(fullR), parameter :: SIX      = 6.0_fullR
        real(fullR), parameter :: SEVEN    = 7.0_fullR
        real(fullR), parameter :: EIGHT    = 8.0_fullR
        real(fullR), parameter :: NINE     = 9.0_fullR
        real(fullR), parameter :: TEN      = 10.0_fullR
        real(fullR), parameter :: TWOTHIRD = 0.666666666666667_fullR
        real(fullR), parameter :: HALF     = 0.500000000000000_fullR
        real(fullR), parameter :: THIRD    = 0.333333333333333_fullR
        real(fullR), parameter :: FOURTH   = 0.250000000000000_fullR
        real(fullR), parameter :: SIXTH    = 0.166666666666667_fullR
        real(fullR), parameter :: EIGHTH   = 0.125000000000000_fullR
        real(fullR), parameter :: NINTH    = 0.111111111111111_fullR
        real(fullR), parameter :: ROOT2    = 1.414213562373095_fullR
        real(fullR), parameter :: ROOT3    = 1.732050807568877_fullR
        real(fullR), parameter :: PI       = 3.141592653589793_fullR
        real(fullR), parameter :: TWOFIVESIX = 256.0_fullR
save
end module type_vars


module tryI

   USE type_vars
   implicit none

  contains

   subroutine try_ia_work(a,size)
   use type_vars
   implicit none
   integer(singI) :: size
   real(fullR),dimension(3,size) :: a
   integer(singI) :: i

   do i=1,size
     a(1,i) = one
     a(2,i) = two
     a(3,i) = three
   enddo

   do i=1,size
     a(3,i) = a(1,i) + a(2,i)
   enddo
   end subroutine try_ia_work

   subroutine try_ia_return(a,size)
   use type_vars
   implicit none
   integer(singI) :: size
   real(fullR),dimension(3,size) :: a
   integer(singI) :: i
```

```
    a(1,1) = one
    a(2,1) = a(1,1) + one
   end subroutine try_ia_return


   subroutine try_ip_work(a,size)
   use type_vars
   implicit none
   integer(singI) :: size
   real(fullR),dimension(:,:),pointer :: a
   integer(singI) :: i

   do i=1,size
     a(1,i) = one
     a(2,i) = two
     a(3,i) = three
   enddo

   do i=1,size
     a(3,i) = a(1,i) + a(2,i)
   enddo
   end subroutine try_ip_work

   subroutine try_ip_return(a,size)
   use type_vars
   implicit none
   integer(singI) :: size
   real(fullR),dimension(:,:),pointer :: a
   integer(singI) :: i

    a(1,1) = one
    a(2,1) = a(1,1) + one
   end subroutine try_ip_return

end module tryI

program main
   USE type_vars
   USE tryI
   implicit none
!
   interface
     subroutine try_up_work(ap,size)
     use type_vars
     implicit none
     integer(singI) :: size
     real(fullR),dimension(:,:),pointer :: ap
     end subroutine  try_up_work

     subroutine try_up_return(ap,size)
     use type_vars
     implicit none
     integer(singI) :: size
     real(fullR),dimension(:,:),pointer :: ap
     end subroutine  try_up_return
   end interface
!
   integer(singI) :: i,ntimes = 1000,            &
                     size = 1000000
!
   real(fullR),dimension(:,:),allocatable :: ax
   real(fullR),dimension(:,:),pointer     :: ap => Null()
   real(fullR)              :: start,finish
   real(fullR),dimension(14):: mtime = zero

!
! Allocatable array
   allocate(ax(3,size))
   ax = zero
```

```
   call cpu_time(start)
   do i=1,ntimes
     call try_ua_work(ax,size)
   enddo
   call cpu_time(finish)
   mtime(1) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ua_return(ax,size)
   enddo
   call cpu_time(finish)
   mtime(2) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ia_work(ax,size)
   enddo
   call cpu_time(finish)
   mtime(3) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ia_return(ax,size)
   enddo
   call cpu_time(finish)
   mtime(4) = finish-start

   deallocate(ax)
!
! Pointer array
   allocate(ap(3,size))
   ap = zero

! Pointer to an uninterfaced array
   call cpu_time(start)
   do i=1,ntimes
     call try_ua_work(ap,size)
   enddo
   call cpu_time(finish)
   mtime(5) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ua_return(ap,size)
   enddo
   call cpu_time(finish)
   mtime(6) = finish-start

! Pointer to an interfaced array
   call cpu_time(start)
   do i=1,ntimes
     call try_ia_work(ap,size)
   enddo
   call cpu_time(finish)
   mtime(7) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ia_return(ap,size)
   enddo
   call cpu_time(finish)
   mtime(8) = finish-start

! Pointer to an uninterfaced array
   call cpu_time(start)
   do i=1,ntimes
     call try_ua_work(ap(1:3,1:size),size)
   enddo
   call cpu_time(finish)
   mtime(9) = finish-start
```

```
   call cpu_time(start)
   do i=1,ntimes
     call try_ua_return(ap(1:3,1:size),size)
   enddo
   call cpu_time(finish)
   mtime(10) = finish-start

! Pointer to an interfaced array
   call cpu_time(start)
   do i=1,ntimes
     call try_ia_work(ap(1:3,1:size),size)
   enddo
   call cpu_time(finish)
   mtime(11) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ia_return(ap(1:3,1:size),size)
   enddo
   call cpu_time(finish)
   mtime(12) = finish-start

! Pointer to an interfaced pointer
   call cpu_time(start)
   do i=1,ntimes
     call try_ip_work(ap,size)
   enddo
   call cpu_time(finish)
   mtime(13) = finish-start

   call cpu_time(start)
   do i=1,ntimes
     call try_ip_return(ap,size)
   enddo
   call cpu_time(finish)
   mtime(14) = finish-start

   deallocate(ap)
!
! Cannot do pointer to pointer without an explicit or implicit interface
!
! Print CPU summary
   print *,"Allocatable arrays"
   print *," CPU time for un-interfaced with work ",mtime(1)
   print *," CPU time for un-interfaced with simple ",mtime(2)
   print *," CPU time for interfaced with work ",mtime(3)
   print *," CPU time for interfaced with simple ",mtime(4)
   print *,"Pointer arrays  pointer-to-allocatable  range specified"
   print *," CPU time for un-interfaced array with work ",mtime(5)
   print *," CPU time for un-interfaced array with simple ",mtime(6)
   print *," CPU time for interfaced array with work ",mtime(7)
   print *," CPU time for interfaced array with simple ",mtime(8)
   print *,"Pointer arrays  pointer-to-allocatable  undefined range "
   print *," CPU time for un-interfaced array with work ",mtime(9)
   print *," CPU time for un-interfaced array with simple ",mtime(10)
   print *," CPU time for interfaced array with work ",mtime(11)
   print *," CPU time for interfaced array with simple ",mtime(12)
   print *,"Pointer arrays  pointer-to-pointer"
   print *," CPU time for interfaced with work ",mtime(13)
   print *," CPU time for interfaced with simple ",mtime(14)
end program main

subroutine try_ua_work(a,size)
   use type_vars
   implicit none
   integer(singI) :: size
   real(fullR),dimension(3,size) :: a
   integer(singI) :: i

   do i=1,size
```

```
  a(1,i) = one
  a(2,i) = two
  a(3,i) = three
enddo
do i=1,size
  a(3,i) = a(1,i) + a(2,i)
enddo

end subroutine try_ua_work

subroutine try_ua_return(a,size)
use type_vars
implicit none
integer(singI) :: size
real(fullR),dimension(3,size) :: a
integer(singI) :: i

 a(1,1) = one
 a(2,1) = a(1,1) + one

end subroutine try_ua_return
```